Jesper Lehtinen

Automated GUI Testing of Game Development Tools

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Degree Programme

Bachelor's Thesis

25 May 2016

Metropolia

Author(s) Title Number of Pages Date	Jesper Lehtinen Automated GUI Testing of Game Development Tools 42 pages 25 May 2016
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Vesa Ollikainen, Senior Lecturer Petteri Salo, Tools Team Manager

The goal of this thesis was to automate testing of the graphical user interfaces of tools used in game development at Remedy Entertainment Ltd. The technologies and methods were chosen in advance, and the choices were made with maintainability, ease of use and meaningful result reporting in mind. A continuous integration system which supported running automated tests already existed, but no actual tests existed yet.

As a result of this thesis, a system, that automatically runs graphical user interface tests whenever code is submitted to the version control system, was built. The tests covered the basic user interface functionality of the game development tools, such as asset manipulation and file input and output. The major part of this thesis was a test application, which was added as a part of the continuous integration system. Adding test application as part of the continuous integration system required additional steps such as resetting up the test environment file structure.

As a conclusion it can be noted that both manual and automated testing have their place within the software development environment and successful teams utilize a combination of both methods. In most cases automating testing is worthwhile when planned carefully. Automated testing should be focused on verifying that already established functionality remains unchanged, and manual testing should focus on finding new defects through exploratory testing.

Keywords	Automated, GUI Testing, UIAutomation

Tekijät(t) Otsikko Sivumäärä Aika	Jesper Lehtinen Automatisoitu pelinkehitystyökalujen käyttöliittymätestaus 42 sivua 25.5.2016
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Vesa Ollikainen, Lehtori Petteri Salo, Tools Team Manager

Tämän opinnäytetyön tavoitteena oli automatisoida pelien kehityksessä käytettävien työkalujen graafisten käyttöliittymien testausta Remedyllä. Teknologiat ja metodit oli valittu ennen työn aloitusta. Valintoihin vaikutti eritoten ylläpidettävyys, käytettävyys ja tulosten raportointi. Jatkuvan integraation järjestelmä oli jo valmiiksi pystyssä, mutta sen testausominaisuuksia ei ollut vielä otettu käyttöön.

Opinnäytetyön tuloksena oli järjestelmä, joka pystyy automaattisesti suorittamaan pelien kehityksessä käytettävien työkalujen graafisten käyttöliittymien testejä aina koodimuutosten tullessa jatkuvan integraation järjestelmään. Testit käsittivät pelien kehitystyökalujen käyttöliittymien peruskäytön, kuten resurssien editointi ja tiedostojen tallennus sekä luku. Työn keskipisteenä oli testisovellus, joka lisättiin osaksi jatkuvan integraation järjestelmää. Testisovelluksenn käyttäminen vaati ylimääräisiä toimenpiteitä, kuten esimerkiksi testausympäristön tiedostorakenteen automaattinen alustus.

Johtopäätöksenä voidaan todeta, että niin ikään manuaalisella kuin automaattisella testauksella on omat paikkansa ohjelmistokehityksessä, ja menestyvät tiimit käyttävät onnistuneesti molempia menetelmiä. Useimmissa tapauksissa automaattinen testaus kannattaa, kunhan se on suunniteltu huolella. Lisäksi automaattisen testauksen tulisi keskittyä olemassaolevan toiminnallisuuden testaukseen. Manuaalinen testaus taas kannattaa keskittää uusien vikojen löytämiseen tutkivamman testauksen menetelmällä.

Avainsanat	Automated, GUI Testing, UIAutomation

Metropolia

Contents

List of Abbreviations

API	Application programming interface.
AUT	Application under test. Refers to the application that is currently undergoing testing.
COM	Component Object Model.
DLL	Dynamic-link library. Microsoft's implementation of a shared library intended to be shared by multiple programs.
GUI	Graphical user interface.
UIA	Microsoft UI Automation library.
MSAA	Microsoft Active Accessibility.
NUnit	NUnit is a unit-testing framework for all .Net languages.
SUT	System under test. Refers to the system that is currently undergoing testing.
WED	WorldEditor is the level editor developed in-house at Remedy.
WPF	Windows Presentation Foundation is a system used for rendering user interfaces using DirectX on Windows-based applications.

1 Introduction

This thesis was done at Remedy Entertainment Ltd., to explore the possibilities of automated testing in game development. The aim was to provide stability through all stages of production in different projects at Remedy. More specifically this thesis focuses on automating testing of the tools that are used for creating games. This should help in creating a more stable production environment as the quality of the tools directly reflect the quality of the actual game.

The number one goal of this thesis was to create an automated graphical user interface (GUI) testing pipeline to aid in the development of the internal game development tools. This consisted of a system that was capable of automatically launching the tool applications, identify different UI elements, and perform various functional tests by interacting with the UI elements. The goal was to not only get this system up and running, but also have it provide useful and relevant information constantly. This meant balancing between test specificity and maintainability. Very specific test cases would give very focused information of the software under test but can be tricky to maintain when the software is constantly changing. Another part of the goal was to find an ideal test frequency as it is not feasible to run every test after every code submit. For this reason, categorization of the tests into more frequent ones and longer ones was needed. The automated test system was integrated as an additional step in the build chain.

Another goal of this thesis was to investigate how automatically testable the tools at Remedy are already as is and what kind of modifications they would require to make automated testing more viable. The in-house tools will also never be finished, but rather keep constantly changing in order to meet different project needs. At Remedy there exists an interesting combination of legacy and modern software. Having software from different eras provided a great opportunity to approach this subject from different angles.

This thesis should benefit anyone who is in the process of setting up an automated test environment from scratch. The thesis covers the up- and downsides of the technologies used, this can help others to evaluate if those technologies fit their testing needs. There are also concrete examples of test cases being used at Remedy, which can be used as reference when creating brand new test cases. The basics of how to do good testing are covered as well, from planning to the actual execution and reporting of the test results.

2 Game Development Tools at Remedy

Remedy is a Finnish game developer known for its various AAA-titles. Currently Remedy employs over 120 employees from various disciplines; ranging from programmers, level designers and 3D-modelers all the way to the screenplay writers and cinematic scripters.

The main focus of Remedy is story-driven third person shooter games, with a heavy focus on the story part. To drive the story Remedy uses vigorous amounts of dialogue, cinematics and written content. The visual look of the games is very often realistic with added supernatural gameplay elements. The Alan Wake and Max Payne -series are good examples of this typical Remedy style as shown in Figure 1.

Figure 1. Screenshot from Remedy's Alan Wake.

In the following a look is taken at how these different disciplines work together and what kind of tools they use to create the style that Remedy is known for.

2.1 Common Game Asset Pipelines

Video games nowadays consist of huge amounts of data in various different formats, from visual to audible and to even haptic feedback. Somehow all this data needs to come together from these different sources to produce an immersive experience. In practice

there are various types of game asset pipelines all requiring multiple steps from various different disciplines, and covering them all would make for a thesis on its own. By looking at a simplified game object pipeline, one can get a good understanding of the workflows and what kind of data there is, which in turn helps to map out the testing needs.

An object in a 3 dimensional game world will most often need some sort of a defined structure. This structure is called a mesh. The mesh data contains a number of points in 3D space, also known as vertices, and triangles which connect the different vertices thus forming the object's surface. The use of triangles in the mesh data is because they are the fastest format to process on the hardware. [1] The amount of these triangles, also known as polygons, that are being rendered is a good way to measure how demanding an object is for the hardware.

In modern game engines how the surface of an object should look is often defined by something called material. A material is often a combination of at least 3 different textures (these are typically images) and typically a shader. These textures are the view-independent texture called the albedo map, which only contains the color information of the material. Then a normal map which contains the lighting information of the material which fakes the bumps and dents on the material. The third most commonly used texture map is the specular map, which is used to tell the game renderer how shiny the different areas of the material are. [2] A shader is a small program that is used for computing shading in runtime.

A basic game object will consist of a mesh and a material. This data will then be used by the game engine's renderer to form a visual presentation of the game object in virtual 3D-space. The different textures will be most often created in a professional photo/image editing software such as Photoshop. Meshes are made in specially tailored 3D modeling software e.g. 3ds Max or Blender. This software is also used in the process of mapping the different texture coordinates onto the meshes, also called texturing.

Single game objects can also consist of multiple meshes. Most often for example any character in the game will have different meshes in a structured hierarchy which makes creating animations for the characters easier.

2.2 Essential Tools at Remedy

The tools at Remedy pick up from where the image editing and 3D modeling tools leave the data at. There is the mesh data and the textures mapped on top of that data, that is something the game engine and the renderer can often use already, but when thousands and thousands of assets exist, there will be a need for metadata to help organize the assets. To construct any sort of virtual world of all these assets, they need to be given a specific transform inside the world and have different behavior scripted in. In most game engines a physics simulation system exists which allows these different objects to interact with each other within set rules. The objects will, in addition to shape and look have physical data linked to them, for example mass, friction and elasticity. The metadata and these additional attributes are the sort of attributes that the tools will be editing.

Next, the three most essential tools used and developed by Remedy are covered. Starting from the very essential level editor, continuing to the material editor and lastly looking at the dialogue editing tool.

2.2.1 WorldEditor

Developed from ground up at Remedy, WorldEditor (WED) is a virtual world editor with multiple different components. It is used mainly for level design, including scripting and landscaping. Level design can be broken down into placing different kinds of game objects in a three-dimensional space and controlling their behaviors through scripts. Figure 2 shows the default layout of the WorldEditor. On the left-hand side there is the “Hierarchy tree” showing the level asset structure.

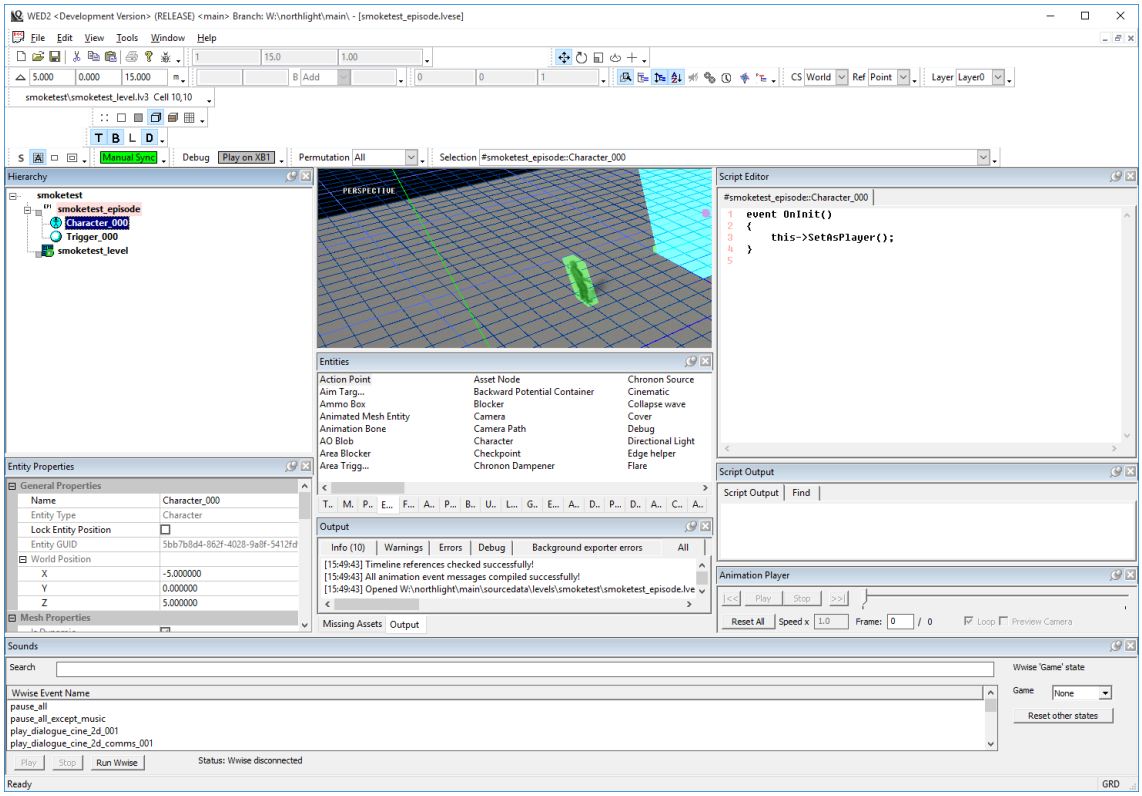

Figure 2. Main user interface of the WorldEditor with the default layout.

One level in WED consists of an episode file and a level, or landscape file. The episode files usually contain more dynamic assets such as characters and game objects that can be interacted with. The level or landscape file contains more static elements such as buildings, trees and terrain. Each entity placed in the level has their own properties which can be seen and edited in the “Entity Properties” –view.

The middle pane contains the editor viewport, which displays the game world with all the added entities and landscape. The game world can be navigated by flying around using a game pad or the keyboard. Automatically testing if the view in the viewport is actually correct can be often tricky. One good example of a difficulty is a comparison of the expected results of a particle effect with randomized particle movement, as it has a different look every time it is simulated.

Under the viewport there is the asset browser pane with multiple different tabs for selecting various different types of assets to be added to the level. The output pane, which provides useful information to the user, is located under the asset browser. This output tab can be utilized in the automated test assertions. On the right-hand side there is the

“Script Editor”. Each of the entities in the level can be manipulated by a scripting language. In Figure 2 the shown script will set the character placed in the level as the player character when the level loads. The “Script Output” pane is under the “Script Editor” pane, providing the user with possible script compilation or other errors. The last two bottom elements are the “Animation Player”, for previewing animations, and the “Sounds” pane for adding audio to the level or entities.

The WorldEditor is an integral part of the typical workflow, it is where all of the content comes together. That is why levels are often edited by multiple people at the same time and WorldEditor needs to be able to handle that.

2.2.2 Butler

Butler is Remedy’s game asset manager and a material editor tool. It allows the user to filter assets by different categories, such as meshes, textures, particles and materials. The preview feature also allows the user to see the changes made instantly in the game. Finding the right look for a material is an artistic process and often requires multiple iterations, so this in-game preview feature speeds up the workflow by a lot.

Butler also serves as an interface to the version control system to help simplify the workflow by marking edited assets automatically as being under edit and blocking submit of assets with missing references. In most cases a missing reference on an asset will either cause a default, very noticeable asset to be used or in the worst case it will cause a crash, stopping others from working.

Figure 3 displays the asset browser and the filtering features, first the assets are filtered to show only textures and additionally files that match the text “checkerboard”. On the right-hand side the properties of the selected asset are shown.

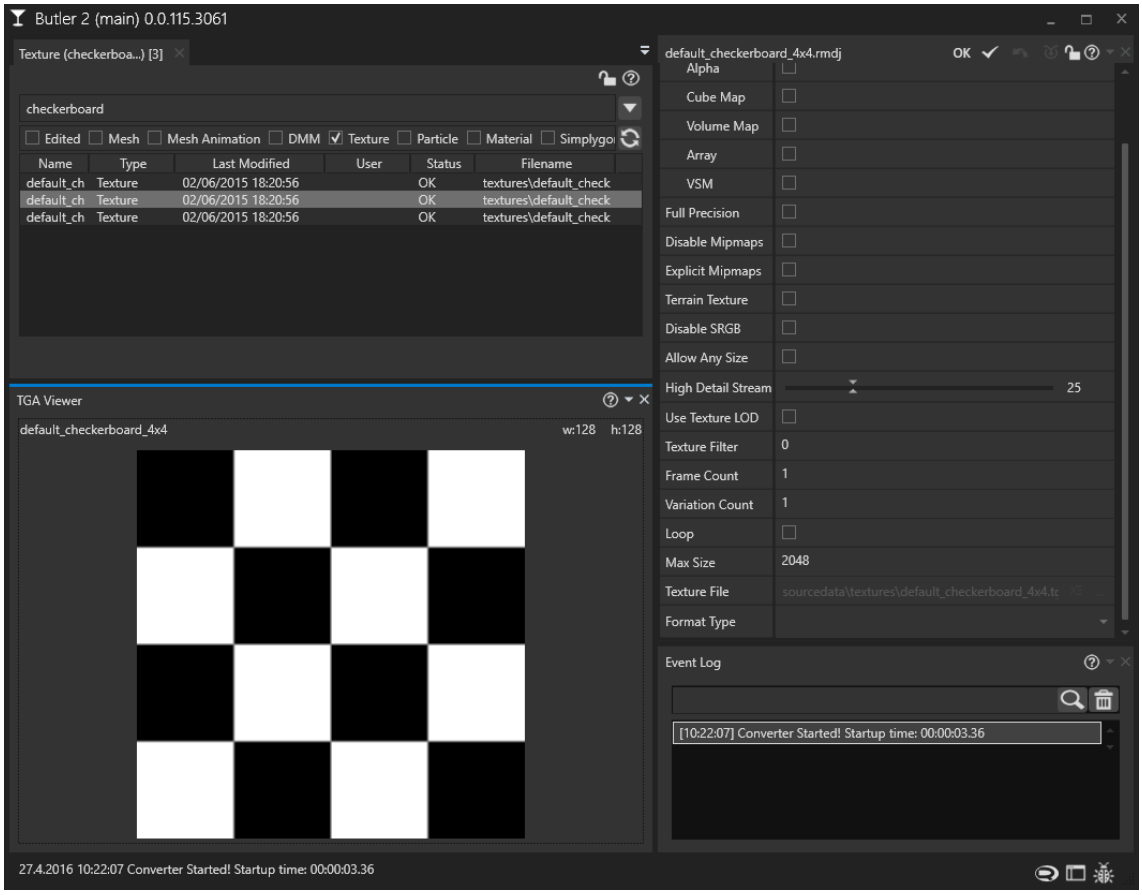

Figure 3. Butler with 4 docked windows.

Butlers UI has been built using the Windows Presentation Framework (WPF) framework thus it supports multiple dockable windows. This means that the UI can dynamically change depending on the use case. That is something that needs to be taken into account when designing the actual test cases for this tool by having a way to define a constant layout for the GUI for each test.

2.2.3 DialogueWriter

As Remedy uses a lot of dialogue in their games to drive the story, a need for a proper tool to manage all that dialogue was necessary, thus DialogueWriter was developed. Aside from being a text editor which is specialized in producing documents in a dialogue format, it also features an audio player. In Figure 4 the audio player is displayed in the top left corner, the text editor on the right side and the dialogue database on the bottom.

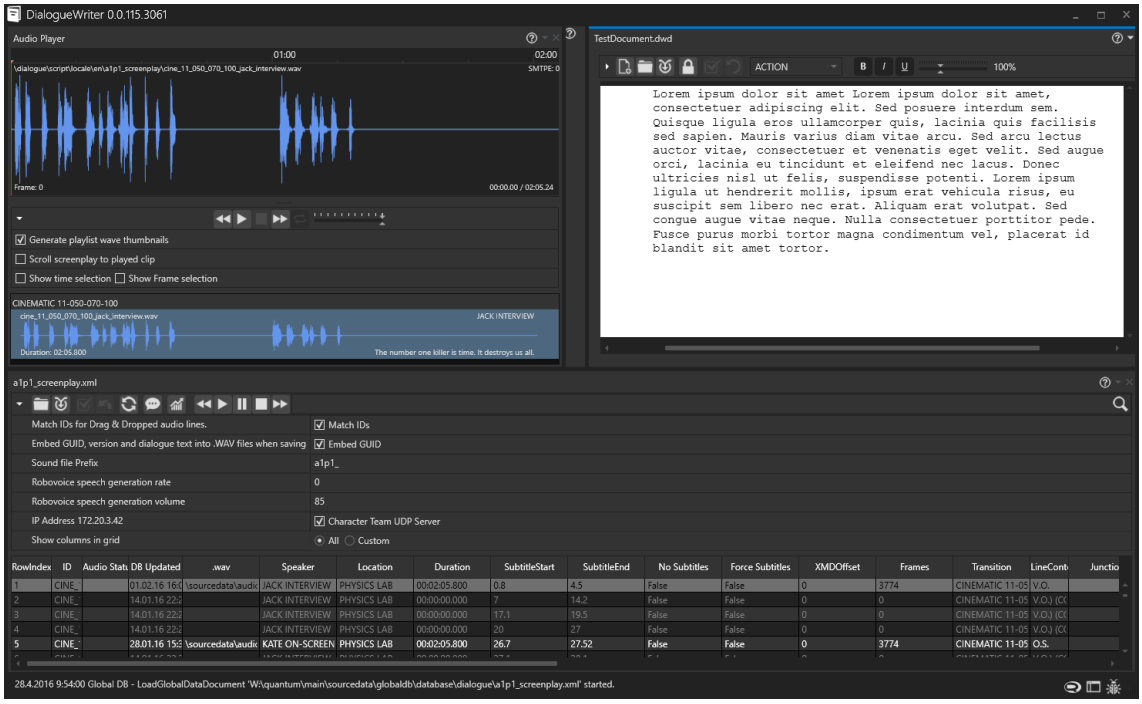

Figure 4. DialogueWriter's UI displaying the audio player, dialogue database and editor view.

The audio player can be used to preview different dialogue clips and allows the text editor view to be synchronized to the clip currently being previewed. The audio player also allows for simultaneous play of different audio clips; which is useful for timing dialogue between two characters.

The dialogue database view shows different metadata regarding the dialogue. It lists information such as the dialogue speaker, location in the level, duration, subtitle timings and other related information. Some of these fields are editable by the user and the changes should reflect the changes to the metadata.

3 Software Testing

Why is software testing necessary in the first place? Why is it not possible to just develop the software as it was designed on paper? The reality is that mistakes are always made, even on the design stage, especially when working under strict deadlines. Software testing exists to primarily serve the software developers to find these mistakes or defects. By running tests (be it manual or automatic), it can be verified that newly added features function the way they are meant to and no unwanted side-effects, also known as bugs, were introduced. The more testing is done, the better is the chance of finding the bugs

early. The earlier these bugs are found, the easier it is to fix them as there are fewer possible causes and developers are still focused in that specific area of the software. Finding a bug that has been lurking in the system for months requires often plenty of investigation and can lead to speculative fixes.

Similarly to software development, software testing is also prone to human error and can at times provide false information on when a bug was first introduced for example. First time a tester spots the bug might not be the first time it has actually occurred. This leads the investigation already to the wrong tracks. Especially manual testing can provide varying results depending on the test at hand and what time of the day it currently is. Automated testing on the other hand can be done more thoroughly, more frequently and faster. When planned carefully it will save significant amounts of time and helps meet the strict deadlines in the first place. Automating the most mundane and important tests will free human testers to focus more on the complex high level functionality, that can not possibly be tested automatically.

The main purpose of software testing is to save time, but testing itself also takes time. Rex Black offers some insight on how to determine the scale of the testing needs in a software project with the following variables and formulas [3, p. 22].

Cost of detection includes all the costs caused by the testing process. All the individual steps such as planning, running the tests and comparing results already produce costs, even if no bugs are found.

Cost of internal failure is part of the costs that actually finding bugs cause. Filing a bug report and verifying a fix are good examples of internal failure costs.

Cost of external failure on the other hand includes the costs of caused by bugs making it to production. For instance, technical support and temporary workarounds are considered costs of external failure. Formula 1 gives a way to estimate the “Average Cost of a Test Bug”, ACTB.

$$ACTB = \frac{cost\ of\ detection + cost\ of\ internal\ failure}{\text{test bugs}} \quad (1)$$

Test bugs in Formula 1 refer to the amount of bugs detected by testing. Basically this metric measures the average effort that is put into detecting, reporting and fixing a single bug.

With Formula 2, ACPB which is an abbreviation of “Average Cost of a Production Bug”, can be estimated. In practice ACPB gives an idea about how much dealing with a bug making its way to production costs the organization. Production bugs signifies the amount of bugs that were not spotted on the testing stage and sneak their way to the production stage.

$$ACPB = \frac{\text{cost of external failure}}{\text{production bugs}} \quad (2)$$

To estimate the return on investment for testing Formula 3 can be used. The formula can be interpreted in a way that the more it costs for the organization to deal with bugs in the production environment, the more should be invested in testing. On the other hand, investing too much into testing can also be detrimental as the average cost of a test bug starts rising.

$$Testing\ ROI = \frac{(\text{ACPB} - \text{ACTB})test\ bugs}{\text{cost of detection}} \quad (3)$$

With the tools at Remedy the production stage is reached when the tools have been compiled and provided for the content developers to work with. The production stage consists of 10 to 70 users depending on the tool. Bugs reaching the production stage in WED are the costliest to deal with, as it is the most used tool at Remedy. Very severe bugs slipping through can cause days of lost work time. Handling a bug in the internal phase costs much less by average than handling it in the production phase. Internal phase can be considered the phase when the feature is still in development and has not been deployed for usage.

Currently at Remedy there are no automated tests running on the build machines and everything is tested manually, from the various tools in use to the actual final game. This is why a return of investment for automated testing can be expected to be very beneficial. Next chapters will cover different test processes and methods to identify areas that should be automated.

3.1 Testing Processes

There are various different ways to approach the process of testing. The processes range from the more informal ones of ad-hoc testing, to the more formal ones with individually defined steps. Fewster & Graham [4, p. 13] describe the following model that is very similar to the traditional waterfall model used in software development. In theory, the execution of any of these steps can be either manual or automatic.

3.1.1 Identifying Test Conditions

This step involves identifying the different parts of the software that can be tested. It is also a good idea to try to categorize the test conditions in some ways.

In Remedy's case the test conditions will be the very basic user interactions with the tools. In WorldEditor those actions are opening, saving and creating levels and episodes, adding entities, modifying their properties and editing scripts. For Butler the chosen test conditions are filtering actions and asset property editing. Property editing itself will include adding components to assets and interacting with sliders and checkboxes and other UI elements.

3.1.2 Designing Test Cases

Designing the test cases is basically determining how the different test conditions will be tested. It could prove useful to separate the test cases into manual and automated in this step, by using the categorization from the first step. Good candidates for automated test cases are the ones that are repeated constantly and have results that can be compared easily by a machine, for example a byte by byte file comparison. When designing the test cases, it is also good to think about who the software is being tested for, be it for example developers or end users.

3.1.3 Building Test Cases

Building up the test cases means setting up the test cases in a way that they can be executed. This means writing a test script, preparing input and other test data together with the expected outcomes. When testing the game development tools, one expected

outcome can be a game asset with a specific set of attributes. The assets produced after executing the test cases will then be later compared to the expected outcomes.

In this thesis the test cases were built in a C# class library project, and the expected outcome assets were created with versions of the tools that were verified to be working.

3.1.4 Executing Test Cases

Executing the test cases is simply following the test scripts as they are written. In manual testing this often means following a written guide on what inputs to provide to the system and what results to expect. In an automated environment this means that a machine is following the test script, and will do exactly as written. Unlike their machine counterparts, human testers are more flexible in situations where a deviation from the test plan is required, this is why automated testing is more fragile.

The execution step is the main focus of the thesis at hand. This means that the interaction with different UI elements in the Remedy tools will be automated. Crash handling is something that also needs to be handled automatically during the test execution phase.

3.1.5 Comparing Results

After executing a test case where an asset is modified with a tool, the attributes of the asset are then compared to an existing asset which was set up earlier during the building phase. An important thing to note is that a machine can only compare, but never verify the results. Verifying will always be done by a human tester. When result comparison shows differences, it is first the responsibility of the tester to verify that the test was ran under the right conditions, that the test environment was set up correctly, the test script is correct and that the expected results are correct. After these steps the result can be deemed to be in fact erroneous. [4, p. 23]

With the Remedy tools the comparison step will be automated when applicable. The more straightforward comparison methods to see if files are identical or if a file exists will be used more. For the visual and aesthetic features, automatic screenshots during testing will be collected and manually verified.

3.2 Testing Methods

Part of developing a successful test process is to choose the correct testing methods. During the planning stage the conditions that can be tested will be determined and then prioritized. After the test conditions have been defined then in turn it should be determined how the conditions will be tested, and in this stage the testing methods and other higher level test case details should be defined.

3.2.1 Black Box Testing

In Black box testing the AUT is treated as a “Black box”, which means there is no visibility on how the input is handled inside the system, only on how it is supposed to be handled. The tester simply provides input to the system and receives output. The focus on the testing is then on the application externals and functional testing. For this reason, automatic dynamic assertions are trickier to handle. For example, asserting automatically that a graphical user interface is showing the right elements is depended on pixel by pixel screenshot comparison.

This approach can be taken to simulate more of the actual end-user experience. The end-user at Remedy is an artist, designer or programmer who uses the provided tools. The black box approach will test more of the functionality, if the system meets business requirements and user acceptance. [5, p. 238]

3.2.2 White Box Testing

The in-house game development tools at Remedy all have their source code readily available, thus the applications can be tested using the so called white box testing method. In white box testing the inner workings of the individual components of the software under test can be monitored. For example, the GUI layer can be isolated from the rest of the software simply to verify that different GUI elements invoke the actions in a correct way without taking into account what the result of the action should be.

In general, white box testing is easier to automate than black box testing, as the focus is on the application internals and there is a direct access to variables and methods. The approach also enables the developers to measure how much of the code is covered by

tests. Bugs found with the White box approach are often easier for the developers to fix as the problem areas can be more often pinpointed more accurately. [5, p. 238]

3.2.3 Grey Box Testing

Someone did go and coin the term “Grey box testing” and it is considered as a combination of black box and white box testing. Even when there is full access to the source code, the software is still approached as a “black box” in some cases. In other cases, it can be much more feasible to access the source code and circumvent areas that cause problems in testing. One example would be internally inserting faulty values and verifying that the UI notifies the user correctly.

4 Automated Software Testing

Noted by Fewster & Graham was that, when testing is automated, it is more precisely one of the testing process steps that is automated. In theory it is possible to automate identification of test conditions, test case design and test case building, but in practice it would need a lot of investment to get usable results. This thesis aims to automate steps where the return of investment is the highest, so the planned system will be both executing the test cases and comparing the results automatically, rest of the steps will still be manual. The approach taken to develop the automated test system was from a perspective of what can go wrong. Idea behind this approach was to avoid common pitfalls and thus create a more useful system.

4.1 Common Problems in Test Automation

The following list of common known problems listed by Fewster & Graham was taken into consideration when building the automated testing pipeline. [4, p. 10]

1. Unrealistic expectations. Once the system is complete, are human testers even needed anymore? With no experience in automated testing, the expectations from the automated system can be quite high. When the actual implementation of the execution automation started, the limitations, but also the power of automated testing were quickly realized.

2. Poor testing practice (automating chaos just gives faster chaos). This point can be seen just slowing down the development of the automated tests, but not really preventing their creation. It would be a great starting point if verbose and specific test cases already existed before the test automation process began. An automated system can be set in place in a way that test cases can be easily added.

3. Expectation that automated tests find a lot of new defects. Most of the bugs are found in explorative testing, which is something where a tester gets to be creative with their testing methods and test limits of the applications. With that said, the expectation that a computer program doing testing would be capable of creativity and imagination can be linked to the first point of unrealistic expectations. Acknowledging this point, the focus of the automated tests should be shifted more towards assuring the existing functionality.

4. False sense of security, can be easily created if the tests are built wrong. To tackle this, care needs to be taken when creating the test cases and the expected outcomes. The test cases should be reviewed also on a regular basis.

5. Maintenance of automated tests, can become a burden if the planning phase is not executed well enough. The maintenance of the tests was taken heavily into consideration when choosing the technologies and tools for testing. Instead of a recorder tool which relies simply on coordinates a more dynamic programmatic access to the UI element was chosen.

Keeping these common problems in mind when designing the system makes the following stages a lot easier.

4.2 Unit Tests

In unit testing a very specific area of code, also called a unit, is tested in a specific context. Most of the time units are synonymous with functions. Unit tests are often written by the developer as the software is developed, side by side. The main point of unit tests is to assure that the developer understands how each of the particular pieces of code are working. Unit testing does not take into account the quality of the code, just that it does what is wanted from it. [6, p. 5]

The current unit test code coverage at Remedy is about 10 percent. The plan in the future is to increase the coverage to reach a percentage closer to 90 and over. As the codebase is fairly old, and has not been programmed with testing in mind, it will take a lot of refactoring to be able to effectively implement unit tests. The automated GUI testing was developed as an additional safety net to guarantee the high level functionality during the refactoring process.

4.2.1 NUnit

NUnit provides a framework to structure test cases and allows dynamic comparisons with the provided assert methods. Both of these features help with maintainability and getting meaningful results from the tests.

The assert methods are a way for use to assure that a condition is true, for example by comparing data bytes. The following are examples of the assert methods are:

```
Assert.Less (x, y)
Assert.Greater (x, y)
Assert.AreEqual(expected, actual [, string message])
Assert.IsTrue(bool condition [, string message])
```

The naming conventions in NUnit are meant to produce readable code, and the methods should do exactly what they are named after. The “Assert.Less” function asserts for example that x is less than y. “Assert.Greater” again asserts that the value x is greater than y and so on. [6, p. 35]

Next a look is taken at a very simple code sample that gives an idea of how the tests can be structured in NUnit.

```
using NUnit.Framework;

[TestFixture]
public class SumTest
{
   [Test]
   public void TestOfSum()
   {
     int a = 1;
     int b = 2;
     int c = 3;
     int sum = a + b + c;
```

```
      Assert.AreEqual(sum, 6);
    }
}
```

To support the testing of multiple different applications in a single project the tests can be structured by the use of an NUnit attribute called “TestFixture”. With the “TestFixture” attribute a class that contains tests can be marked and it allows the usage of the optional setup and tear down methods. The methods to be tested are marked with the NUnit “Test” attribute. The NUnit runner executes each of these methods sequentially. [7]

All failures in the assertions are reported through the NUnit test runner. Whenever an error is encountered, the current test under execution is aborted and the test runner starts the execution of the next test. [6]

4.3 Integration Tests

After the different units have been successfully tested, the units are then tested on how they work together. This step will always assume that the unit tests have been successfully passed, and is taken into account in the test case design.

4.4 Automated GUI Testing

Automated GUI testing can be considered the highest level of testing, as it is the closest to what the end-user will experience. Invoking an action on an UI element will most often end up covering multiple different code paths. When GUI testing is executed by using the Black box method, the testing fully simulates end user experience.

Creating automated GUI test cases can be generally divided into two categories. One approach is to use a recorder tool which records the user’s actions on the desktop and can then be played back at a later date. This approach is easy and quick for simple test case generation, but poses a bigger problem with maintainability. The recorded test case does not take into account if the UI layout of the application under test (AUT) has changed, and with software that is constantly changing this can often be the case. After a layout change, every test would need to be rerecorded.

Another approach is to have some sort of application programming interface (API) to programmatically invoke actions on the AUT. Adding this sort of system to software would mean exposing lot of the functionality to a layer where other programs can access it.

What framework or tool to use is heavily dictated by the platform that the target applications are running on. Selenium for example is one of the more popular GUI testing tools for automating browser based applications. It offers both a recording/playback tool and a scripting language for programmatic access. Selenium supports writing test scripts with the major programming languages like C#, Java and Python. As the internal test framework the test scripts utilize NUnit when using C# and JUnit, which NUnit is based on, when using Java. [8]

The target applications at Remedy are all desktop applications running on the Windows platform, so the tool of choice was TestStack's White, which only supports programmatic access to UI elements. The used technologies will be covered in the following chapters.

4.4.1 Microsoft UIAutomation Framework

Microsoft has been developing a framework where applications developed for the Windows platform by default expose the UI layer to be programmatically accessible. It was first started as part of the Microsoft Active Accessibility (MSAA) project. The main goal behind MSAA was to provide the means to improve how different programs and operating systems work with accessibility aids, such as the magnifier and text to speech. [9] This framework matches the automated testing needs at Remedy, as all the testable applications are running on the Windows platform. Additionally, the UIAutomation framework is readily available without any further costs.

The Microsoft UIAutomation framework is designed for C/C++ developers and provides an easy access to UI elements on the Windows platform. It utilizes the Component Object Model (COM) objects and interfaces. The UIAutomation framework supports applications based on WPF, Win32 and WinForms. [10] The framework consists of 4 major components. Figure 5 illustrates the relationships and hierarchy of these different components. The provider API which includes definitions for objects that are responsible for providing information about different UI elements and also responding to programmatic input. The client API defines the different types for the client to query information about

the UI and send input to different elements. In an automated testing case the client application is the application that will be performing the tests and reporting the results. The third component is the UI Automation core which provides the implementation for the communication between providers and clients. The UIAutomation proxy elements are a way to communicate with the MSAA server and the User32 and ComCTRL dynamic link libraries, which are Windows operating system components used for controlling and building user interfaces. [11]

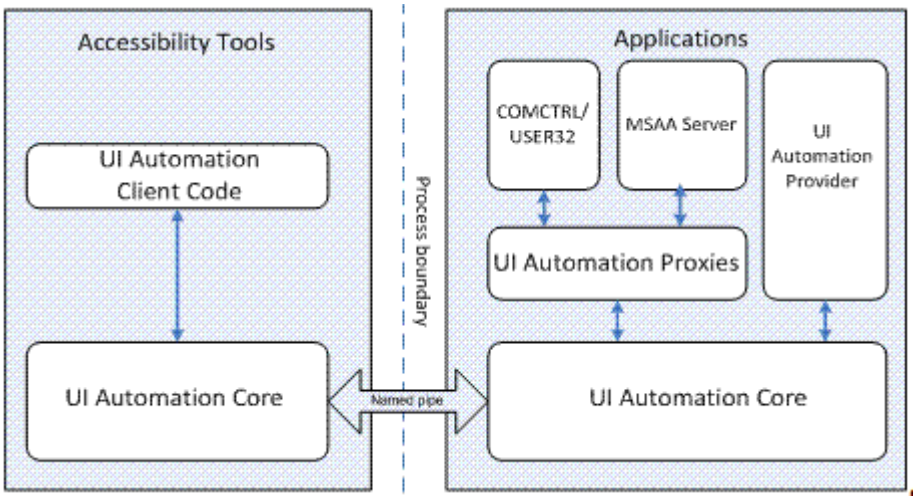

Figure 5. The architecture of the UI Automation framework.

In .NET version 3.0 Microsoft released a managed wrapper around version 2 of the COM-based API. After that version 3 of the UIA framework was released, but the managed wrapper in .NET was not updated. A new open source project of the wrapper was later authored by Michael Bernstein with the name “UI Automation COM-to.NET Adapter” which wrapped version 3, but it is not officially supported by Microsoft thus not part of the .NET distributions. [12]

Microsoft has provided a couple of tools which provide details of the user interface on any application ran on a Windows desktop. Inspect is the most recent one of these tools and it uses UIA version 3 to query for the UI elements.

Figure 6 shows the UI of the calculator from Windows 10 in programmer mode. The Hexadecimal input mode is currently selected and Inspect is set to watch the cursor position.

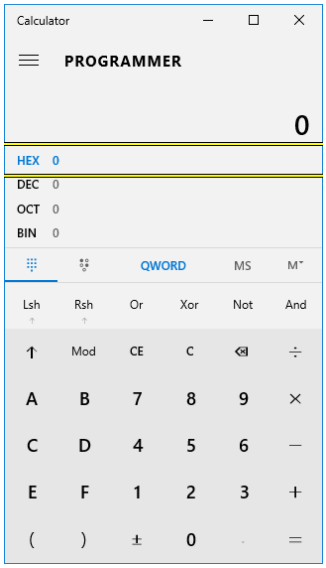

Figure 6. Microsoft calculator with the Hex button focused.

Figure 7 shows the information that the UIAutomation framework provides through Inspect. With these different attributes, specific UI elements can be programmatically queried and applicable actions invoked. The UI elements are arranged in a hierarchy tree with the root element always being the desktop. This structure is important to keep in mind when scoping the element queries.

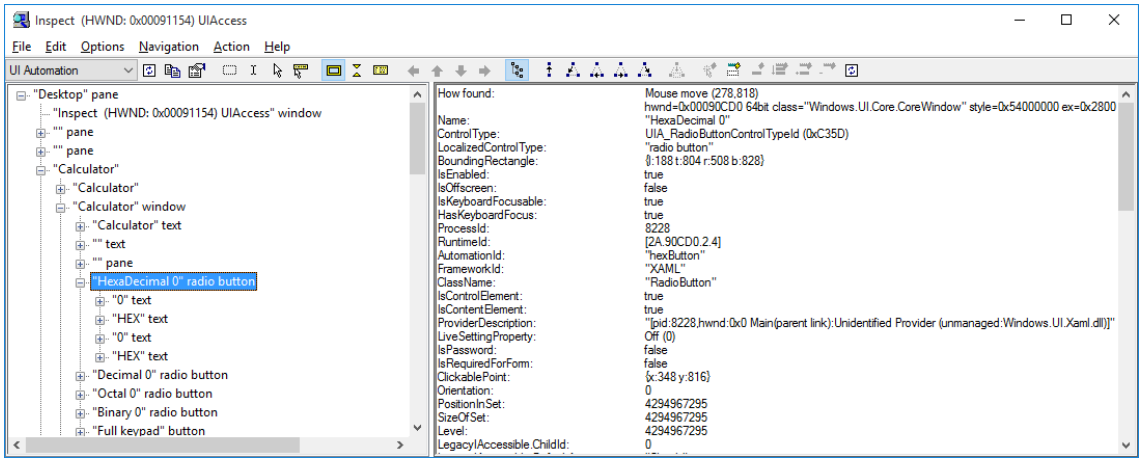

Figure 7. Microsoft Inspect showing the selected button details.

To programmatically access these elements in C#, the appropriate libraries need to be added to the project and a new UIAutomation object created.

```
var UIAutomation = new CUIAutomation();
```

The UIAutomation object contains the root element, which is used as a starting point for the UI element queries. In most cases the root element of the UIAutomation object is simply the desktop pane.

```
var rootElement = UIAutomation.GetRootElement();
```

With the calculator application running it can be seen that the application has a window titled Calculator, so the following query fetches a handle to that window:

```
var windowElement = rootElement.FindFirst(
UIAutomationClient.TreeScope.TreeScope_Children,
UIAutomation.CreatePropertyCondition(
UIAutomationClient.UIA_PropertyIds.UIA_NamePropertyId, "Calculator" )
);
```

The “FindFirst” method returns the first automation element that matches the provided search criteria. Here the parameters are the scope to search in, set to be only the child elements of the window element, then a condition where the name field of the automation element needs to match the string “Calculator”.

To get a handle on the hex button the information that inspect provided in Figure 7 is used. Querying elements under the calculator window happens with the following lines of code:

```
var hexButton = windowElement.FindFirst(
UIAutomationClient.TreeScope.TreeScope_Children,
UIAutomation.CreatePropertyCondition(
UIAutomationClient.UIA_PropertyIds.UIA_AutomationIdPropertyId,
"hexButton" ) );
```

This query is similar to the previous one, except this time the window element is used as the root of the query and the search property condition requires the automation id field to match the string “hexButton”.

Finally, to invoke a click on the hex button the current control pattern of the button is required.

```
( (IUIAutomationInvokePattern)hexButton.GetCurrentPattern(
UIA_PatternIds.UIA_InvokePatternId ) ).Invoke();
```

The line of code above would effectively invoke a mouse click on the hex button. The next chapter shows how these actions can be achieved with a framework built on top of UIAutomation.

4.4.2 TestStack White

White is an open source project GUI automation framework taken over by TestStack. In its current master branch White uses the UIA version 2 and the adaption of version 3 is underway, but to adapt it fully requires major architectural rework. The main goal of White is to provide access to the UIAutomation framework in a more user friendly way, by hiding complexity. The object model in White is strongly typed in comparison to UIAutomation, where most of the time interaction happens with the “AutomationElement” interface. Additionally, White is already integrated with NUnit, so it is better suited towards automated testing. [13]

Next the above example is looked at a new but this time implemented with the White syntax. To start things of the application is launched with the following line of code:

```
Application application = Application.Launch(“Calculator.exe”);
```

A handle to the application process is kept and used to get handle to the main window.

```
Window mainWindow = application.GetWindow( "Calculator" );
```

The “GetWindow” method simply searches for a window which has a title that matches the given string parameter and returns a handle to it. Once a handle to the window exists, a reference to the hex button can be fetched and a click action invoked on it as shown in the code sample below.

```
mainWindow.Get<Button>( ( SearchCriteria.ByAutomationId( "hexButton" )
) ).Click();
```

Comparing the syntaxes between the two examples should illustrate the work that goes under the hood in White. The work put into White makes the common actions required in automated testing more accessible and produces more readable test scripts.

5 Continuous Integration and Delivery

Continuous integration is the practice of compiling and testing the applications on every code submit. The main goal of continuous integration is that the software is in a working state at all times. [14, p. 55]

Continuous delivery means constantly releasing software builds that meet the production stage standards. The number one reason why continuous delivery exists is to make the feedback loop from the end user back to the developer quicker. One good metric here is to measure the time it takes for a single code change to make it into production, also known as the cycle time. [14, p. 138]

In the games industry, it is crucial to have a very short cycle time as game development is a highly iterative process. Game designers need to try out their ideas in a preliminary version of the game to see what works and what does not, what feels dull and what feels exciting. Same goes for the artists, for example the brick wall that was just created needs to be tweaked so it looks amazing in both the lit and unlit areas of the game. These preliminary versions of the game should be as stable as possible, void of crashes and bugs, as they heavily hinder the ability to get feedback by breaking the flow every so often. This sort of feedback loop is one of the basic processes of creating a great game.

5.1 TeamCity

TeamCity is a build management and continuous delivery system. It supports a lot of different environments, from the major issue trackers to the major version control systems and test runners. The basic architecture of TeamCity consists of a central server which communicates with multiple build agents. The server is responsible for monitoring the version control system and allocating different tasks (compiling code, running tests and so on) to the agents. TeamCity supports the NUnit test runner out of the box, which enforced the choice of using NUnit as a test framework even more.

A basic workflow in TeamCity consists of the following steps:

1. The central server detects a change in the version control system and stores it in the database

2. A database trigger then adds a build to the queue.

3. The server finds an applicable agent and assigns the task to it.

4. The agent executes the pre-determined build steps and reports progress back to the server, including log messages, test reports and code coverage results.

5. After the build is done the build agent sends the resulting data (executables) back to the server. [15]

Knowing the basics of the TeamCity pipeline the implementation of the GUI testing pipeline can move forward. The test application that handles the test input needs to be assigned for execution on an agent automatically. An agent also handles the test environment setup as a separate step from the test application.

6 Implementation

Next a look is taken at how the automated testing system of the game development tools at Remedy shaped up to be. The requirements were to have a program that could be ran as part of the continuous integration system as frequently as possible and provide meaningful results. Before starting the implementation, some of the test cases had to be designed as none existed. A choice was made to design test cases that will only cover the very basic functionalities of the tools and proceed forward from that in the future. Creating new, saving, opening, editing and opening of files were considered to be the basic functionality and the test cases revolved around those actions.

As the GUI tests use keyboard and mouse for input, the tests could only be run on an unlocked desktop. This poses a security risk when running unsupervised tests overnight for example. A workaround for this is to run the tests on virtual machines where the

desktop is not locked and then remote in to start the tests. This method was not utilized in this thesis, but is something planned in the future development.

6.1 Test Application Implementation

First a C# project was set up in Visual Studio 2012, the project type was set to be a class library. A class library fit the needs as the produced DLL file was only used by the NUnit test runner.

Shown in Table 1 are the assembly references that were added to the project:

Table 1. Assembly references if the test application.

Name	**Description**
TestStack.White	The White framework. Providing access to Whites implementation of UI items and querying.
Castle.Core	A White dependency. White utilizes the Castle assembly for logging purposes.
NUnit.framework	The NUnit framework provides test management and assertion functionality for us.
UIAComWrapper	Open source version of the managed wrapper for the UIAutomation framework (version 3). This was required so it was possible to query some of the integral UI elements in the tools.
System.Drawing	Provides screen capture functionality.
System.Threading	For utilizing the sleep timers, used in cases when the wait functionality from White wasn't reliable enough.
WindowsBase	Needed for the Point implementation which were used in some cases as mouse click coordinates.

At this point the building of the automated test cases was started. The first step that was automated in the testing process was the test input, interaction with the actual tool UI. One thing to remember is that, developing automated tests is also software developing, so good programming principles apply. The NUnit features were utilized to develop a structure which supports testing multiple different applications within one project. As the

number of test cases grows in the future, it might be a good idea to create a separate project for each of the tools. With the use of NUnits test fixtures different string parameters can be defined to be used in the tests. [7] This helps reduce code duplication by setting up the same initialization steps for each of the fixtures. For example, each of the test cases require a handle to the application and its main window. The base class where the tool specific test classes were derived from is called “ToolsTests”.

```
namespace ToolsTests
{
 public class ToolsTests
 {
  private readonly string toolsFolderPath;
  private ProcessStartInfo processStartInfo;
 protected Application application;
 protected Window mainWindow;

public ToolsTests(string branch, string toolFolderName, string
toolFileName)
{
 toolsFolderPath = Path.Combine(@"W:\northlight", branch, "latest");
 var toolFolderPath = Path.Combine(toolsFolderPath, toolFolderName);
 var toolPath = Path.Combine(toolFolderPath, toolFileName);

 processStartInfo = new ProcessStartInfo();
 processStartInfo.WorkingDirectory = toolFolderPath;
 processStartInfo.FileName = toolPath;
}
```

The base class constructor handles setting up the paths to the actual executable and to the working directory. Additionally, there is the choice to define a version control branch in which the tests are being run. “ProcessStartInfo” is a class from System.Diagnostics which is helpful for programmatically starting processes with different parameters on Windows. Next in the same base class there is the setup method.

```
[SetUp]
public void TestSetUp()
{

 // Start the application and sleep so we have time to init the
windows
 application = Application.Launch(processStartInfo);
 Thread.Sleep(3000);

 // We are expecting the main window to be the first window, index 0
 mainWindow =
application.GetWindow(SearchCriteria.ByControlType(ControlType.Window)
.AndIndex(0), InitializeOption.NoCache);
 mainWindow.WaitWhileBusy();
 mainWindow.Focus();
```

```
 // Keyboard shortcut to maximize the window
 mainWindow.Keyboard.HoldKey(KeyboardInput.SpecialKeys.LWIN);
 mainWindow.Keyboard.PressSpecialKey(KeyboardInput.SpecialKeys.UP);
 mainWindow.Keyboard.LeaveKey(KeyboardInput.SpecialKeys.LWIN);
}
```

First of all, the method is marked with the “SetUp” attribute from NUnit which causes the method to be called before every test that is being run from that fixture. The method will use the “ProcessStartInfo” object that was set up in the constructor and use a method from the White framework to launch an application with the given parameters. Then the program sleeps for 3 seconds to wait for the main window to initialize and gets a handle to it. Last step in the setup method is to maximize the main window by sending a combination of Windows key and the UP arrow key. This is done to have more consistency in the UI layout, which is important when the tests are being run automatically without supervision. The way White provides the keyboard input has had ease of use in mind. Different windows of the application can be targeted separately and special keys on the keyboard can be held down.

Next up is the tear down method responsible for cleaning up the test environment.

```
[TearDown]
public void TestTearDown()
{
 if (TestContext.CurrentContext.Result.Status == TestStatus.Failed)
 {
  Desktop.TakeScreenshot(TestContext.CurrentContext.Test.Name+".png",
System.Drawing.Imaging.ImageFormat.Png);
 }

 if (application != null) application.Kill();
}
```

It was marked with the “TearDown” attribute from NUnit which causes this method to be executed at the end of each test. In its current state the method takes a screenshot if NUnit reports that the test has failed. The method also forces the application to close if the shutdown has not been successful yet.

From the base class the tool specific classes were derived and they included the actual test cases. As an example, a simple test case of launching WorldEditor, opening a level and exporting that level to the game is taken a look at.

For the first step of launching the application the base class needs to be supplied with the correct path to the executables. This is achieved by using the “TestFixture” attribute and providing it with the correct parameters.

```
[TestFixture("main", "tools\\worldeditor", "WorldEditor.exe")]
 public class WorldEditorTests : ToolsTests
 {
  public WorldEditorTests(string branch, string toolFolderName, string
toolFileName)
 : base(branch, toolFolderName, toolFileName)
 {
  CoreAppXmlConfiguration.Instance.MaxElementSearchDepth = 5;
 }
```

Only one instance of the class marked with the “TestFixture” attribute will be created when running the tests, so the constructor is also a good place for application specific configuration settings. For WorldEditor the “MaxElementSearchDepth” was set to five to reduce time it takes to query for different UI elements and have the tests run quicker. The “CoreAppXMLConfiguration” is a configuration file provided by the White framework, and additionally it contains settings for wait and timeout times.

Once the functionality to launch the application to be tested was implemented, the automation of the actual test case input was implemented inside the test fixture. Each of these test methods will be called once preceded by the setup method and followed by the tear down method.

```
[Test]
[Category( "Long" )]
public void OpenLevelAndExport()
{
            …
```

Different test categories can also be utilized in NUnit by using the “Category” attribute and then let the NUnit test runner know which categories to include or exclude in testing. The sample test case has been categorized as long as the export process is often a time consuming process.

At this point the test application and its initialization steps were set up, as the AUT is launched, a reference to the started process is kept and a reference to the main window of the process is fetched. The next step was to use the Inspect tool to find controls of interest. Figure 8 displays the visible hierarchy of the WorldEditor UI as seen by Inspect.

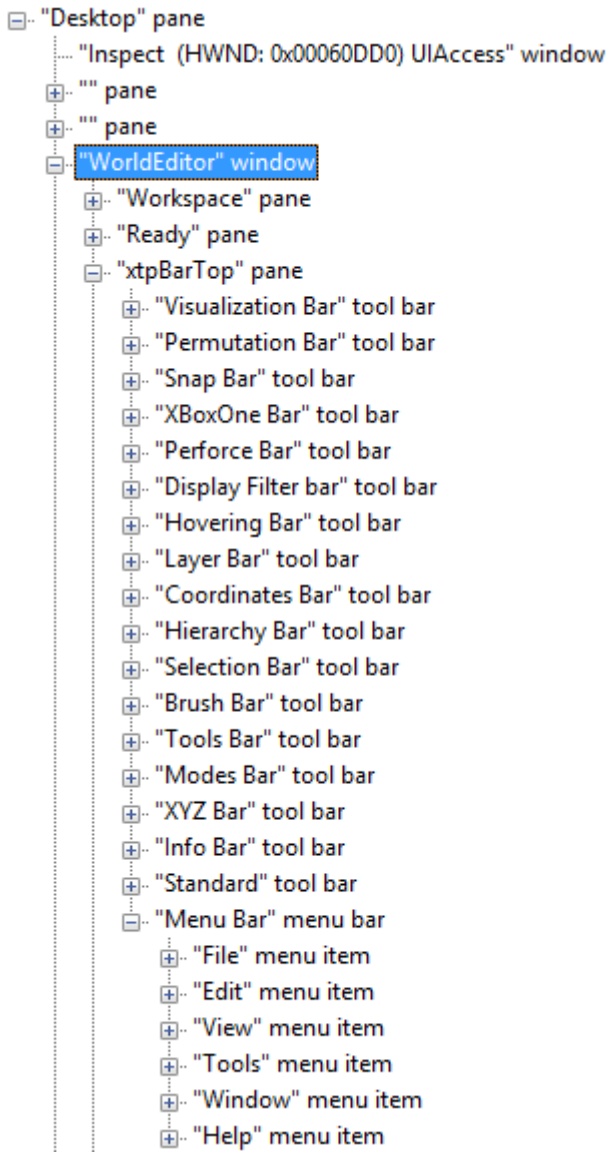

Figure 8. The WorldEditor UI as seen by Inspect.

As the first step in the test case was to open up a level, the file menu and its child components were the first elements to be interacted with. Something to note here is that the child elements are not instantiated until the file menu has been interacted with, usually by invoking a mouse click on the item. Before it was possible to invoke a click action on the file menu, a reference to the “MenuBar” was necessary. The “MenuBar” contains a menu item called “File” which is the common file menu a click action can be invoked on.

```
MenuBar menuBar = mainWindow.GetMenuBar(SearchCriteria.ByText("Menu
Bar"));
Menu fileMenu = menuBar.MenuItem("File");
fileMenu.Click();
```

The click method is provided by White and is shared by every UI item there is. It simply gets the bounds of the UI element in question and invokes a mouse click in the center of those bounds. After the interaction with the file menu, the child elements get instantiated. Inspect recognizes that WorldEditor's main window now contains these new menu items as seen in Figure 9.

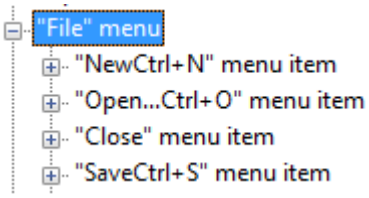

Figure 9. WorldEditor file menu as seen by Inspect.

Here is something to be aware of about the UI hierarchy structure, the "File" menu is not actually instantiated as a child element of the file menu item which was the child of the menu bar, but rather as a child of the main window. Thus the following line of code has to be used:

```
mainWindow.Get<Menu>((SearchCriteria.ByText("Open... Ctrl+O")))
.Click();
```

This queries for a Menu item with the name "Open... Ctrl+O" which is a child of the main window and invokes a click on the center the item, if one is found. In cases where White is unable to find an UI element it retries multiple times before timing out and throwing out an exception. The time it takes to time out can be configured in the White configurations.

After the "Open... Ctrl+O" menu item was clicked, a dialog with the title "Open" will pop up. It is illustrated in Figure 10. From this dialog the level named "smoketest_level" will be opened.

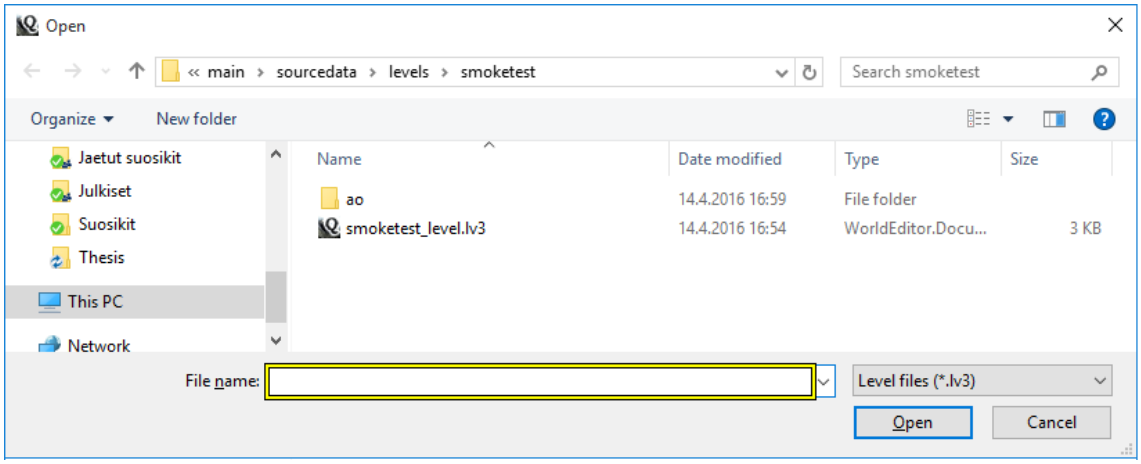

Figure 10. The WorldEditor Open dialog.

There were options of either getting a list of UI items that the dialog provides and clicking on one of them, or interact with the “File name” text edit bar. Using the “File name” text edit bar was chosen, as it can be provided with the full path to the file in question. This choice was made to have more consistent results with the tests.

Figure 11 shows the details of the text edit bar. There were multiple ways of interacting with this specific text edit bar.

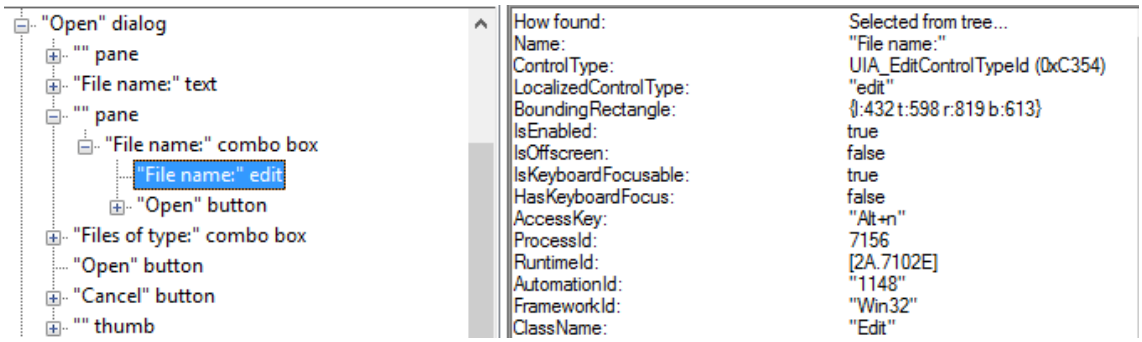

Figure 11. The open dialog as seen by Inspect.

The “AccessKey” field shows that the element can be focused with the combination of the ALT and N keys. Alternatively, the element can be fetched with the name “File name:”, but for the sake of execution time and consistency the keyboard shortcuts were used.

The “ModalWindow” function accesses dialogs that are children of the main window. By using the reference to the open dialog, keyboard actions can be invoked on that specific

window. The keyboard functions provided by White were used to simulate holding down the ALT key and pressing the N key while ALT is still being held down.

```
Window openWindow = mainWindow.ModalWindow("Open");

openWindow.Keyboard.HoldKey(KeyboardInput.SpecialKeys.ALT);
openWindow.Keyboard.Enter("n");
openWindow.Keyboard.LeaveKey(KeyboardInput.SpecialKeys.ALT);

openWindow.Keyboard.Enter(@"…\levels\smoketest\smoketest_level.lv3");
openWindow.Keyboard.PressSpecialKey(KeyboardInput.SpecialKeys.RETURN);

mainWindow.WaitWhileBusy();
```

Then assuming the “File name:” field has the focus, the full path to the level to open was entered and confirmed by simulating a return keypress. After opening the level, it was necessary to wait for WorldEditor to finish loading by using the wait functionality from White. Under the hood, White wait functionality constantly polls the application to see if it is ready to accept user input again.

Once the level was loaded the file menu was interacted with and queried for the “Export to Game...” menu item to invoke a click action on it.

```
fileMenu.Click();
mainWindow.Get<Menu>(SearchCriteria.ByText("Export to Game..."))
.Click();
```

This opened another dialog with the title “Export to Game”, and it was accessed the same way as the “Open” dialog by using the “ModalWindow” function. To start the export process, a button labeled “Start” was queried for, and a click action was invoked on it.

```
Window exportWindow = mainWindow.ModalWindow("Export to Game");
exportWindow.Get<Button>(SearchCriteria.ByText("Start")).Click();

Label exportStatusLabel =
exportWindow.Get<Label>(SearchCriteria.ByAutomationId("1035"));

while(!exportStatusLabel.Text.Contains("finished"))
  {
   Thread.Sleep( 2000 );
  }

 exportWindow.Get<Button>(SearchCriteria.ByText("Close")).Click();
 fileMenu.Click();
 mainWindow.Get<Menu>(SearchCriteria.ByText("Exit")).Click();
}
```

Once the export process had started, a reference to the export status label was fetched, using the automation id the found with Inspect. The automation id is shown the same way for this label as for the edit box which can be seen on Figure 11. After having found that the status label contains the string “finished”, it was determined that the export has completed and the dialog was closed. Then it was just a matter of interacting with the file menu again and selecting “Exit” to close WorldEditor.

To handle crashes during these tests a simple exit code handler was added to the White framework:

```
private void ProcessOnExit( object sender, System.EventArgs e )
 {
   if( process.ExitCode != 0 )
   {
     throw new WhiteException( "Process exited unsuccessfully with
exit code: " + process.ExitCode);
    }
 }
```

When launching the application, the method is assigned to be called on the exit event with the following lines of code:

```
process.EnableRaisingEvents = true;
process.Exited += new EventHandler( ProcessOnExit );
```

With this, White causes the test to fail when an erroneous exit code is detected.

6.2 TeamCity Implementation

Now that test application was set up and there were test cases to run, it was time to integrate the it as part of the build delivery system. For the sake of this thesis a test project was created in TeamCity with the default settings and the version control system was added to the project. The design was to run the tests as an additional build step after each code submit that TeamCity detects. To have a clean slate for the tests that manipulate files, the folder structure was reset to a known state before running any tests.

Inside the test project there was a build configuration called “TestBuild” with the default settings.

The build configuration consists of different settings and it also includes the build steps to be executed when the build is triggered, either manually or automatically. As was discussed earlier, the tests should run as frequently as possible to catch bugs early. That's why it would be ideal to run the tests after every code submit, but this requires the code to be compiled every time, which also takes time and resources. TeamCity handles the resources by queuing the compilation requests, and having the code submits to pile up. The testing process starts after the applications to be tested have been compiled with the new code changes. Figure 12 illustrates how the continuous delivery pipeline has been constructed at Remedy, and where the automated tests are fitted in.

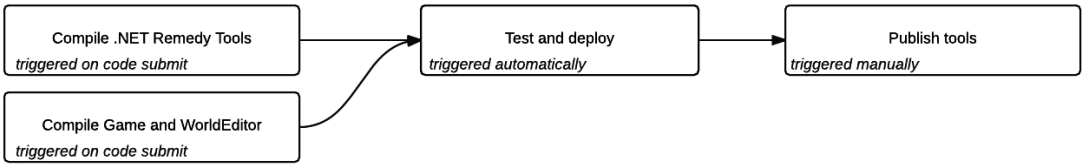

Figure 12. High level illustration of the continuous integration pipeline.

As can be seen from the figure, the tools applications are compiled when TeamCity detects a code submit in the version control system. The test and deploy step gets triggered once both of its dependencies, in this case the compilation of the .NET tools and game and WED executables, are successful. The last step, which is to publish the newly compiled executables and send out an e-mail to the project teams with the release notes, can only be triggered manually and once the dependencies have completed successfully.

The actual test build configuration ended up being quite simple as can be seen in Figure 13. The first step to execute in the build was chosen to be the test environment set up. The second step was to use the NUnit test runner targeting the test application DLL.

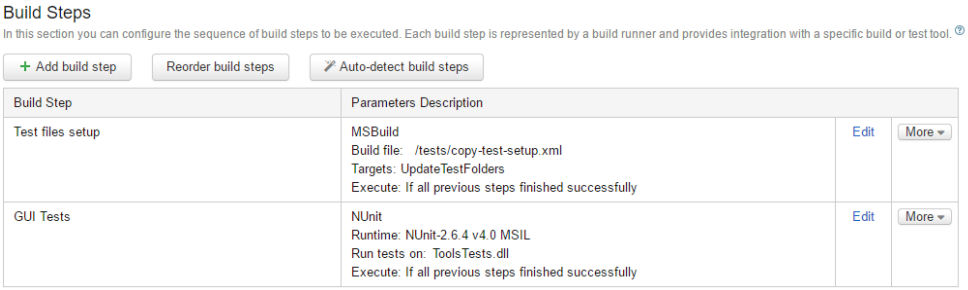

Figure 13. A screenshot from the team city build step manager.

In the setup phase, the folder structure which includes the test files was set up. The expected results will need to be reset and the actual results from the previous tests need to be cleaned. This is done so that the tests are run in a well-known state, thus the output of the different actions that are about to be performed can be predicted. It also keeps the test scripts simpler, as there is no need to add any sort of conditionals to react to different scenarios. For example, when saving a file, a warning pop-up will only appear when a file overwrite is about to happen. By resetting the test environment every time (and verify it being successful), it is known that no other file with the same name exists. The reasoning for running the setup/clean-up step before and not after the tests was that in failure cases investigations of the test execution was still possible.

The setup phase was done by using Robocopy and the mirror option. Robocopy is a file management utility tool developed by Microsoft. The usage of Robocopy from the command line is simple and uses the following syntax:

```
robocopy <Source> <Destination> [<File>[ ...]] [<Options>]
```

The source folder is the root of pre-configured folder structure, and the destination will be a completely copy of it, which will be only used for a single test run and results comparison. The mirror option causes Robocopy to copy all the subdirectories, even if empty, overwrite all files existing files and delete any files that no longer exist in the source. [16]

There is a slight problem with just running the Robocopy step as a command line step inside of TeamCity, and that has to do with exit codes and how TeamCity interprets the exit codes when it comes to step success. TeamCity will fail any build step that reports an exit code that differs from 0, and Robocopy outputs exit codes different from zero even on successful runs. Robocopy will only return the exit code 0 if the no file alterations were made and the program ran successfully. That's why Robocopy runs inside an

MSBuild script that includes a Robocopy extension. The extension handles the exit code problem, makes it compatible with TeamCity, and still reports the erroneous cases correctly. Below in Figure 14 the test file structure for the DialogueWriter tests is shown as an example.

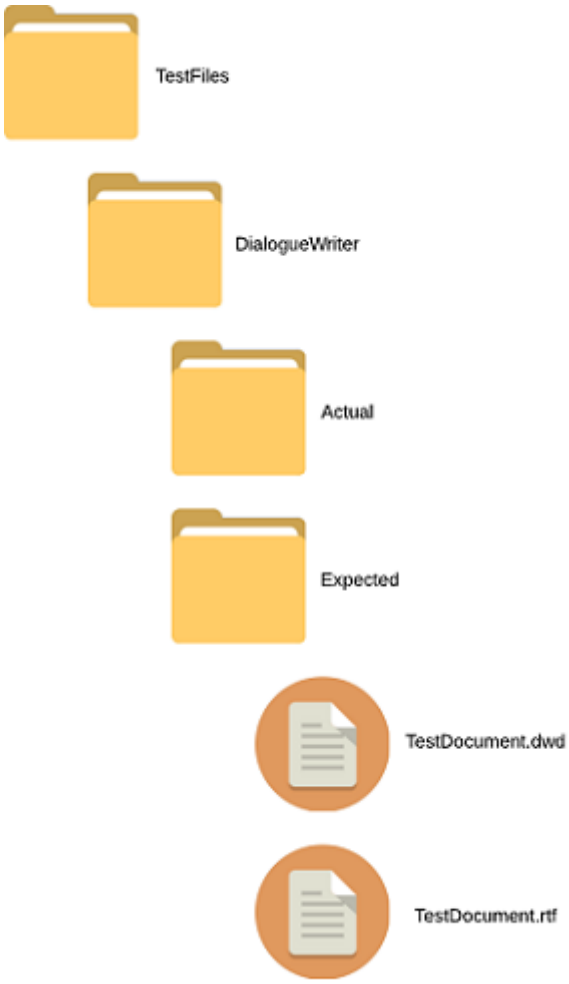

Figure 14. Test file structure for DialogueWriter tests.

When testing the creation of a new document and saving it on the disk, the resulting file was saved under the actual folder and then compared to the file in the expected folder. For the comparison the NUnit file assert function was used. Something to note about the file assert function is that it compares the files byte for byte and fails the test if the files do not match.

```
FileAssert.AreEqual(@"C:\TestFiles\DialogueWriter\Expected\TestDocumen
t.rtf", @"C:\TestFiles\DialogueWriter\Actual\TestDocument.rtf");
```

This was one way of automating the result comparison, and for documents such as this the byte for byte comparison is applicable. There are cases where a unique ID or a timestamp is written inside the file, causing the byte for byte comparison to fail every time even when the all of the content turns out to be correct.

Figure 15 shows the build logs received from TeamCity. It will directly print out the test and test fixture names, so the naming conventions are important when viewing the test results.

Figure 15. Build log from TeamCity providing info on failed tests.

Failed tests are highlighted in red and the output is directly from White. In Figure 15 it can be seen that a DialogueWriter test called “CreateEditAndSaveDocument” failed to find the modal windows called “Save As”. In order to determine what went wrong, the test environment and script need to be verified. When they are deemed to be correct, then a bug most likely exists within the AUT. In this case there were problems with the UI initialization steps, so interaction with the UI elements did not invoke the save functionality.

7 Evaluation

The premise of this thesis was to build an automated GUI testing system for the in-house game developments tools that were tested manually. The technologies to be used were already decided upon based and the choices were based on the platform, test case maintainability and return of investment. As a result of the thesis, an automated testing system was built that can handle the input of the most mundane of test cases. Also some

of the more basic comparison steps were automated. The system was also readied to be implemented as part of the continuous integration cycle.

The UIAutomation framework provides good means to automate UI interaction. Without any rework on the existing applications most of the basic interactability was already available. Much more robust and maintainable automated testing can be achieved, if the UIAutomation framework is taken into account when developing the applications.

With the newest version of the UIAutomation framework there were issues with how the UI hierarchy structure was detected. For example, UIAutomation was only able to detect the checkboxes in the property editor hierarchy in Butler, without any naming labels. Programmatic interaction with the with the checkboxes was possible, but only by index. This made it impossible to design a test case where a specifically labeled check box was clicked.

White was a great addition on top of the UIAutomation framework as it provided easy to use functions and made the common functionality much more accessible. White also provided the integration with NUnit which turned the UIAutomation framework into an automated test framework. These were two of the biggest benefits of using White.

Setting up the tests to run in TeamCity was very straightforward as most of the functionality was provided out of the box. It was simply a matter of pointing the test runner to the correct DLL and setting up the test environment through console commands close to any other environment.

Overall the technologies are quite suitable for testing the Remedy in-house tools. With these technologies at least the very basic functionality can be covered automatically and continuously after every code submit.

8 Discussion

Prototyping first started with TestStack White version 0.13.3 by grabbing the latest package with the NuGet package manager. After a while, some quick test cases were set up for the .NET tools which were using the WPF framework. Only problems that were run into, were UI elements with no unique identifiers, so the test cases had to rely on fixed

UI layout as some elements were queried only by using indexes. When building the WorldEditor test cases, there were problems programmatically querying some of the elements with White. On the other hand, the elements were visible in Inspect and it was possible to invoke actions on the elements through Inspect.

After researching the issue, it was discovered that the UIAutomation framework that Inspect used was of newer version than what White had implemented. The UIAutomation framework is originally intended for C/C++ users and the .NET wrapper has been left without updates after its initial release. Unfortunately, White still uses the outdated wrapper which uses UIA version 2 API. An open source project that wraps the newer and improved UIA version 3 API was later completed, but a version of White that uses it is still in development.

With UIA version 2 it was possible to find all the UI elements on the screen from the WPF based tools. However, with UIA version 2 it was not possible to fetch UI elements from the legacy Win32 based applications. UIA version 3 is able to fetch UI elements from both WPF and Win32 based applications.

The UIAutomation versioning is not well documented on any specific site. The author was lucky enough to stumble upon the versioning history on a blog post by a Microsoft employee discussing how to get started with UIAutomation. The documentation on White also looks extensive at first, but discrepancies are soon noticed here and there. The interest in open source projects amongst volunteers seems to be shifting from being vigorously active to completely disappearing for months. This causes the documentation and features to stagnate, and sometimes outdated documentation is worse than no documentation at all as it leads the reader to the wrong direction.

With access to the source codes, more time could have been spent investigating how the user interfaces of the Remedy tools were constructed. Then more tips on how the user interface should be structured to aid automated testing could have been provided.

9 Summary and Conclusions

The thesis covered a basic game object pipeline in a 3rd person game and example tools used in that pipeline. Once grasping the basic understanding on what the tools do and

why do they exist, the concepts were approached from the perspective of testing. A general test process and most relevant test methods were covered as an introduction to testing. The test process steps to be automated were chosen to be the UI interaction and result comparison as they yielded the best return of investment. The actual implementation of the automation of these steps produced a C# class library which included the test scripts that launches the AUT and executes a set of UI interactions and compares the result to the expected outcome. The test application was also integrated as part of a continuous delivery system.

All in all, implementing the GUI testing pipeline project taught a lot about automated testing. Even with the preparation for all the pitfalls, multiple minor problems were encountered in the actual implementation of the test scripts, most of them being process handling and timing related. Such as invalid crashes caused by killing an application after a test when the process still had work to do. Another issue to take into account was huge variance in application initialization times which were related to caching. Additionally, the framework used to automate testing had some bugs which had to be circumvented. It was also noted that automated testing is not the answer for everything. It outperforms manual testing in tedious and repetitive testing as the machine executes tests with meticulous precision every time.

There are now multiple different directions that the project can be taken to. After covering the very basic functionalities of the tools, more complex test cases can be developed. Those cases might require some UI rework, or additional automation Ids, but will be worth the effort. Getting the tests also to run on virtual machines, would minimize the security risks of having to run the tests on unlocked desktops.

References

1 Ben Carter. 2004. The Game Asset Pipeline. Charles River Media, Inc.

2 Russell, Eddie. 2014. Understanding the Difference between texture maps. Web document. http://blog.digitaltutors.com/understanding-difference-texture-maps/. Read on 27.4.2016.

3 Riley, Tim & Goucher, Adam. 2009. Beautiful Testing: Leading Professionals Reveal How They Improve Software. O'Reilly.

4 Fewster, Mark & Graham, Dorothy. 1999. Software Test Automation: Effective use of test execution tools. Addison-Wesley.

5 Dustin, Elfriede, Rashka, Jeff & Paul, John. 2008. Automated Software Testing: Introduction, Management, and Peformance. Addison-Wesley.

6 Hunt, Andrew. Thomas, David. Hargett, Matt. 2007. Pragmatic Unit Testing: In C# with NUnit. Pragmatic Bookshelf.

7 Maddock, Chris. TestFixture Attribute. Web document. https://github.com/nunit/docs/wiki/TestFixture-Attribute. Read on 10.4.2016.

8 Selenium Project. 2016. Selenium 1 (Selenium RC). Web document. http://www.seleniumhq.org/docs/05_selenium_rc.jsp#introduction Read on 8.5.2016.

9 Microsoft Corporation. August 2001. Web document. https://msdn.microsoft.com/en-us/library/ms971350.aspx. Read on 19.4.2016.

10 Windows Automation API Overview. 2016. Web document. https://msdn.microsoft.com/en-us/library/windows/desktop/dd561932(v=vs.85).aspx. Read on 4.4.2016.

11 Architecture and Interoperability. 2016. Web document. https://msdn.microsoft.com/en-us/library/windows/desktop/dd561882(v=vs.85).aspx. Read on 4.4.2016.

12 Barker, Guy. 2015. So how will you help people work with text? Part 1: Introduction. Web document. https://blogs.msdn.microsoft.com/winuiautomation/2015/09/29/so-how-will-you-help-people-work-with-text-part-1-introduction/. Read on 6.4.2016.

13 TestStack White documentation. How White works? Web Document. http://teststackwhite.readthedocs.io/en/latest/ Read on 7.4.2016.

14 Humble, Jez & Farley, David. 2013. Continuous Delivery: Reliable Software releases through build, test, and deployment automation. Addison-Wesley.

15 Alexandrova, Julia. 2016. Continuous Integration with TeamCity Web document. https://confluence.jetbrains.com/display/TCD9/Continuous+Integration+with+TeamCity Read on 22.4.2016.

16 Microsoft. 2012. Robocopy. Web document. https://technet.microsoft.com/en-us/library/cc733145(v=ws.11).aspx Read on 25.4.2016.